nice to know

CAS Front-End Engineering

JS CLOSURES, MODULES, INHERITANCE

ADVANCED TOPICS

Silvan Gehrig

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

NESTED SCOPES
CLOSURES

BASICS: JAVASCRIPT

Scopes: Behind the Scene

- **Variables (with their states) are bound to the functions**
- **This binding is applied during the function call creation**

```
function myFunction() {
    var inner = 5;
    var getInner = function() {

        return inner;

    };
    return getInner;
}
```

Scope contains a list of referenced variables and referenced Scopes

Returns the execution information for function getInner

Scopes: Behind the Scenes: Execution Context

```
function myFunction(arg) {
    var inner = arg;
    var getInner = function() {
        return inner;
    };
    return getInner;
}
( myFunction(6) )(); // 6
( myFunction(7) )(); // 7
```

Variables Object (or Activation Object) [VA]

- Data of the Execution Context (Variables / Function Declarations / Parameters)

Scope

- Chain with parent Variable Objects for identifiers lookup

Returns the whole execution information *VA* + *Scope* for function getInner

- **The binding of variables is applied during entering an execution context**
- **The active execution context is created at function call**
- **Source: http://www.ecma-international.org/ecma-262/5.1/#sec-10.3**

OOP WITH JAVASCRIPT

Which technics are needed for OOP?

- **Classes**
  - Properties
  - Methods
- **Polymorphism (Inheritance)**
  - Method/Operator Overloading
  - Method Overriding
- **Modules / Namespaces**

OOP BASED API

OOP WITH JS

Operators

- **new Operator**
  - Creates a new Object Instance
  - Changes the Context to the newly created Object during the creation process
  - Constructor Functions can return an object; default value is the created object
  - Example

```
function House () { }
let houseObject = new House();
```

- **instanceof Operator**
  - Tests if the given object is derived from another
  - To lookup is done by visiting the inheritance (prototype) chain
  - Example

```
houseObject instanceof House // returns true
houseObject instanceof Object // returns true
```

Object Creation

- **Object Literal**

```
let newObject = { color: "green", colorize: function() { } };
```

- **Object Function**

```
let newObject = new Object();
newObject.color = "green";
newObject.colorize = function() { };
```

- **Object.create Static Method**
  - Directly inherits from the given prototype (applies the prototype inheritance)

```
HighRise.prototype = Object.create(House.prototype); // use create() instead of new House();
```

- **Constructor Function / Class Definition**

OOP based Properties

- **Function.prototype**
  - Specifies the inheritance chain (as object)
  - Every function/constructor has a writeable prototype property

- **Object.prototype.constructor**
  - Returns the Function that created the instance
  - Can’t be set on read-only or native constructors (e.g. primitive types)
  - Has no affect on the Script runtime
    - Setting the constructor is useful if you use plain Objects as Classes
  - Example

```
function House () { }
let houseObject = new House()
houseObject.constructor; // points to function House
```

OOP based Functions / Methods I

- **Object.prototype.valueOf()**
  - Used when Unboxing a Reference to a Primitive Type
  - Automatically called by the Script Runtime when applying operators
  - Should always return a Primitive Type

```
function House(height) {
    this.height = height;
    this.valueOf = function() { return height; };
}
(new House(100) > 120)                    // results in false
```

- **Object.prototype.toString()**
  - Returns the String representation of the current Object
  - The default implementation returns "[object Object]"
  - Can be overwritten to return something like `[object ${this.constructor.name}]`

OOP based Functions / Methods II

- **Function.prototype.call(thisArg [, arg1[, arg2[, ...]]])**
  - Changes the context to execute the current function on the context thisArg
  - Similar to

```
function myMethod(arg1, arg2, arg3, arg4) { }
thisObj.myMethodOnCtx = myMethod;
thisObj.myMethodOnCtx(arg1, arg2, arg3 , arg4);
```

- **Function.prototype.apply(thisArg, [argsArray])**
  - Changes the context to execute the function on the context thisArg by passing arguments as an array
  - Similar to

```
function myMethod(arg1, arg2, arg3 , arg4) { }
thisObj.myMethodOnCtx = myMethod;
thisObj.myMethodOnCtx( … [ arg1, arg2, arg3, arg4 ] );
```

LANGUAGE BASED CONSTRUCTS

OOP WITH JS

PROTOTYPAL INHERITANCE

OOP WITH JS

Prototypal inheritance - Example

```
function House() {
    this.color = “red”;
}
// House object contains a “static” property called prototype (given from Function)


var redHouse = new House();
// redHouse.__proto__ points to the House.prototype

House.prototype.colorize = function(newColor) { };   // extend existing prototype of class House

redHouse.colorize(“green”);                          // works due the lookup to the __proto__
```

Chaining Classes with Prototype Inheritance

- **If the class's prototype has also a __proto__ property**
  - The lookup is extended to this object
- **This allows to create inheritance hierarchies**

```
function House() { }
House.prototype.setAddress = function() { };

function HighRise() { }
HighRise.prototype = new House();          // HighRise.prototype.__proto__
                                           // Remarks: Sample shows ES4 approach
                                           // in ES5 use Object.create(House.prototype)
var empireStateBuilding = new HighRise();
empireStateBuilding.setAddress("350 5th Ave"); // empireStateBuilding.__proto__
```

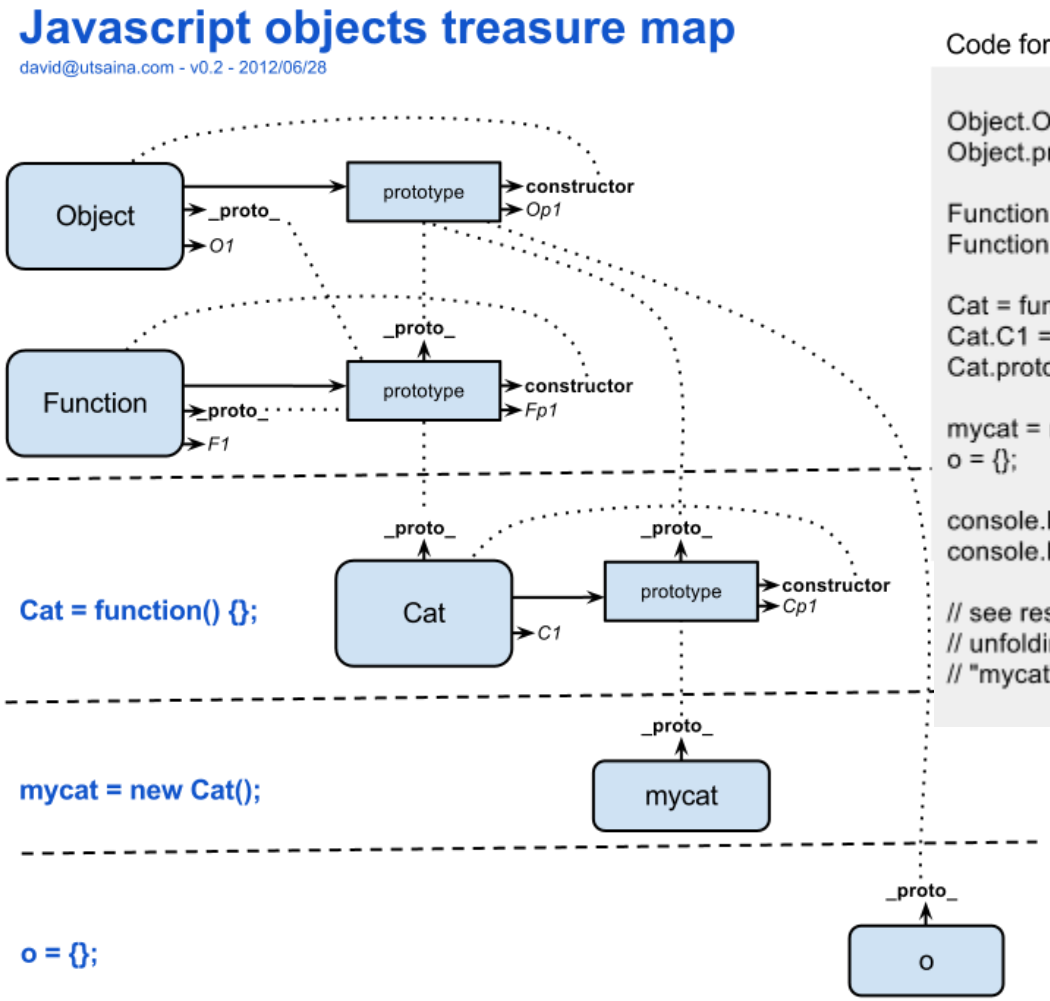
Prototypal inheritance

- **Every function has a writeable prototype property**
- **This prototype contains an internal object from which instances inherit**
- **Allows multiple instances of an object to share a common property**
- **Every object instance of a contains a reference to the prototype of the construction Class**
  - This property is called __proto__
  - __proto__ is standardized in ES2015
- **The property lookup is extended to the __proto__ property**

More detailed information can be found in *Speaking JavaScript*, chapter *The Prototype Relationship Between Objects*

Prototypal inheritance illustrated

reference



Code for testing the map

```
Object.O1='';
Object.prototype.Op1='';

Function.F1 = '';
Function.prototype.Fp1 = '';

Cat = function(){};
Cat.C1 = '';
Cat.prototype.Cp1 = '';

mycat = new Cat();
o = {};

console.log(mycat);
console.log(o);

// see result i.e. in Chrome console,
// unfolding the links of the
// "mycat" and "o" objects
```

Code for creating prototype chain

```
// example: derive Cat from Object (implicitly chained)
Cat = function() { };
Cat.prototype = Obejct.create(Object.prototype);
Cat.prototype.constructor = Cat;
```

Code for creating prototype chain with Animal parent class

```
// example: derive Cat from Animal
Animal = function() { };
Cat = function() { };
Cat.prototype = Obejct.create(Animal.prototype);
Cat.prototype.constructor = Cat;
```

More detailed information can be found in *Speaking JavaScript*, chapter *The Prototype Relationship Between Objects*

Prototypal inheritance - Conclusion

- **Be aware of the length of the prototype chains**
- **Avoid extending the native prototypes (unless for compatibility issues)**
- **Problematic when creating class hierarchies**
  - **Properties are referenced by the *prototype*, NOT re-constructed**
  - ***prototype* property is statically bound the Function Objects – it's for all object instances the same**
  - **If class hierarchies are required, Constructor Inheritance should be used**

CONSTRUCTOR INHERITANCE

OOP WITH JS

Constructor inheritance

- **Base constructor is called in the context of the derived type**
- **Properties with referenced objects are allocated in the constructor**

this = [object HighRise]

```
function House() {

    this.facade = new Facade();

}
function HighRise(address) {
    House.call(this);                    // call function House in context of the new object
    this.facade.setStreet(address.street);
}

let empireStateBuilding = new HighRise( { street="350 5th Ave" } );
let oneWtc = new HighRise( { street="285 Fulton St" } );
```

COMBINING INHERITANCE

BASICS: JAVASCRIPT

Mixing concepts Scope/Context in ES5 I

```
function House (color) {                        // class definition, constructor function
    var self = this;                            // store this pointer on the scope
    self.facadeColor = color;
    self.paint = function(newColor) {           // method definition
        self.facadeColor = newColor;            // do more paint stuff here, colorize windows, etc…
    };
}

var whiteHouse = new House("white");
whiteHouse.paint("beige");                      // call method directly on object; works in any case
var paintWhiteHouse = whiteHouse.paint;         // copy pointer of function paint
paintWhiteHouse();                              // call function without object; works now
```

Mixing concepts Scope/Context in ES5 II

- **Caution: Side effects**
  - JavaScript default behavior overridden

```
var whiteHouse = new House("white");
var paintWhiteHouse = whiteHouse.paint;          // copy pointer of function paint
var aNewObject = {
    "facadeColor": "red",
    "paint" : paintWhiteHouse
};                                               // copy pointer of function paint
aNewObject.paint();                              // paint() is executed on Object whiteHouse
```

Type Conversion / ES5 Constructor Functions

- **Not natively implemented**
- **Calling the primitive value constructors without new() converts a value**
  var num = Number("3");
- **Can be implemented in your own constructor's logic**

Example: Inheritance/Class Transcompilation (TypeScript 1.x)

- **Best of both worlds – mixing the concepts (constructor & prototype)**
  - **Provides** instanceof **operators**
  - **Provides the appropriate** constructor **property**
  - **No problems with same references on instances**
  - **An example provides TypeScript inheritance**

```
var __extends = this.__extends || function (d, b) {
    for (var p in b) if (b.hasOwnProperty(p)) d[p] = b[p];
    function __() { this.constructor = d; }
    __.prototype = b.prototype;
    d.prototype = new __();
};
var Animal = (function () {
    function Animal(name) {
        this.name = name;
    }
    Animal.prototype.move = function (meters) {
        alert(this.name + " moved " + meters + "m.");
    };
    return Animal;
})();
var Snake = (function (_super) {
    __extends(Snake, _super);
    function Snake(name) {
        _super.call(this, name);
    }
    Snake.prototype.move = function () {
        alert("Slithering...");
        _super.prototype.move.call(this, 5);
    };
    return Snake;
})(Animal);
```

Prototype Inheritance

Class Animal

Class Snake

Constructor Inheritance

Example: Class with ES5 - Scope/Context Combined Approach

```
function House (color) {                    // class definition, construc
    var self = this;                        // store context on the scope
    var height = 0;                         // private field definition (similar to private int height;)
    self.facadeColor = color;               // public property definition (similar to public Color facadeColor;)

    self.paint = function(newColor) {       // public method definition
        repaint(newColor);
    };
    function repaint(newColor) {            // private method definition
        self.facadeColor = newColor;
    }
    Object.defineProperty(self, 'height', {     // property definition with accessors
        get: function() { return height; },
        set: function(value) { height = Number(value); }
    } );
}
```

```
class House {        // Java Syntax
    public House(Color c) { facadeColor = c; }
    private int height;
    public Color facadeColor;

    public paint(Color newColor) {
            repaint(newColor);
    }
    private repaint(Color newColor) {
            facadeColor = newColor;
    }

    public int getHeight() { … }
    public void setHeight(int height) { … }
}
```

ES5 Inheritance – Conculsion I

- **In ES5 no real class definition was given**
- **Functions were invoked by using the new operator**
- **“Real” OO-inheritance must be implemented by the programmer**
  - **Existing Prototypal inheritance wasn’t sufficient**
  - **More appropriate mechanism needed, called Constructor inheritance**

- **Constructor inheritance must be combined with the Prototype inheritance**
  - **Otherwise some language constructs such as instanceof operators or duck-typing rules break**

- **Sophisticated inheritance algorithms required to combine both approaches**

ES5 Inheritance – Conculsion II

- **OOP and JavaScript (ES5) don't mix well**
  - Multiple approaches available
  - Every framework defines its own OOP restrictions
  - Complexity of OOP designed JavaScript (ES5) classes may overwhelm the project's complexity

As a recommendation you should use pre-compilers such TypeScript if

«you're working in a team larger than 5 developers

… or have to write more than 10 Script files»

Johannes Rieken, Microsoft

Class System with ES2015 (ES6)

- **New class syntax were introduced with ES2015**
- **Mechanism to derive a class from another**

Today, there's no need to implement the inheritance by yourself

If you need ES2015 features in older browsers (such as Internet Explorer 11), use cross-compilation (e.g. Babel)

…or use TypeScript which provides more powerful typing mechanisms and compiles classes to ES5

- …but also comes with a prototype/constructor based Polyfill for inheritance mechanism

Class System with ES2015: Hoisting Behaviour

- **No hoisting for classes available**
  - **extends** derives from a parent class
  - Can also be used to invoke «mix-ins»
  - This prevents classes from hoisting
    - Mix-in's must be executed at the class definition

PATTERNS AND IDIOMS

Idiom: Immediately-invoked Function Expression (jQuery)

- **Usages**
  - jQuery Scope

```
;(function($) {
    // closure scope, do your stuff here
}(jQuery));
```

  - Passing window and document

```
;(function(window, document) {
    // window, document are directly accessible
}(window, document));
```

Source: http://javascript.crockford.com/code.html

Idiom: Namespace

```
;(function(namespace) {
    'use strict';

    // your code goes here
    // namespace.method = function() { };
})(window.namespace = window.namespace || { } );
```

- **Intent**
  - Avoid collisions with other objects or variables in the global namespace.
  - Packages the code into easily manageable groups that can be uniquely identified.

Source: https://packagecontrol.io/packages/JavaScript%20Patterns

Pattern: Revealing Module I

```
var moduleName = (function() {
    var privateVar;
    function privateFunction(arg1, arg2) {
        // method body
    }
    function publicMember(arg1, arg2) {
        // method body
    }
    return {
        member: publicMember
    };
}()); // use it with moduleName.member()
```

Pattern: Revealing Module II

- **Intent**
  - Slightly improved Module Pattern
  - Variables / functions existence is limited to within the module's closure
  - More fragile than those created with the original Module pattern
    - Replacing public members results in unpredictable behavior

Source: http://addyosmani.com/resources/essentialjsdesignpatterns/book/

Pattern: Singleton

```
let singletonName = (function() {
    let instance;
    function init() {                    // initialize the singleton; module pattern implementation
        let privateVar = 5;
        return {                         // return public variables & functions
        };
    }
    return {
        getInstance: function() {
            if (!instance) { instance = init(); }
            return instance;
        }
    };
}()); // use it with singletonName.getInstance()
```

Pattern: Singleton II

- **Intent**
  - Restricts instantiation of a class to a single object
  - Provides a single point of access for functions
  - Are lazily initialized to reduce startup time

Source: http://addyosmani.com/resources/essentialjsdesignpatterns/book/